A very fast string matching algorithm for small alphabets and long patterns (Extended abstract)

Christian Charras[1], Thierry Lecroq[1], and Joseph Daniel Pehoushek[2]

[1] LIR (Laboratoire d'Informatique de Rouen) and ABISS (Atelier Biologie Informatique Statistique et Socio-Linguistique), Faculté des Sciences et des Techniques, Université de Rouen, 76128 Mont Saint-Aignan Cedex, France.
{Christian.Charras,Thierry.Lecroq}@dir.univ-rouen.fr ***
[2] JDPeh@aol.com

Abstract. We are interested in the exact string matching problem which consists of searching for all the occurrences of a pattern of length m in a text of length n. Both the pattern and the text are built over an alphabet Σ of size σ. We present three versions of an exact string matching algorithm. They use a new shifting technique. The first version is straightforward and easy to implement. The second version is linear in the worst case, an improvement over the first. The main result is the third algorithm. It is very fast in practice for small alphabet and long patterns. Asymptotically, it performs $O(\log_\sigma m(m + n/(m - \log_\sigma m)))$ inspections of text symbols in the average case. This compares favorably with many other string searching algorithms.

1 Introduction

Pattern matching is a basic problem in computer science. The performance of many programs is determined by the work required to match patterns, most notably in the areas of text processing, speech recognition, information retrieval, and computational biology. The kind of pattern matching discussed in this paper is exact string matching.

String matching is a special case of pattern matching where the pattern is described by a finite sequence of symbols. It consists of finding one or more generally all the occurrences of a pattern $x = x_0x_1 \cdots x_{m-1}$ of length m in a text $y = y_0y_1 \cdots y_{n-1}$ of length n. Both x and y are built over the same alphabet Σ.

String matching algorithms use a "window" that shifts through the text, comparing the contents of the window with the pattern; after each comparison, the window is shifted some distance over the text. Specifically, the window has the same length as the pattern x. It is first aligned with the left end of the text y, then the string matching algorithm tries to match the symbols of the pattern

*** The work of these authors was partially supported by the project "Informatique et Génomes" of the french CNRS.

with the symbols in the window (this specific work is called an *attempt*). After each attempt, the window *shifts* to the right over the text, until passing the end of the text. A string matching algorithm is then a succession of attempts and shifts.

The aim of a good algorithm is to minimize the work done during each attempt and to maximize the length of the shifts. After positioning the window, the algorithm tries to quickly determine if the pattern occurs in the window. To decide this, most string matching algorithms have a preprocessing phase, during which a data structure, z, is constructed. z is usually proportional to the length of the pattern, and its details vary in different algorithms. The structure of z defines many characteristics of the search phase.

Numerous solutions to string matching problem have been designed (see [3] and [10]). The two most famous are the Knuth-Morris-Pratt algorithm [5] and the Boyer-Moore algorithm [1].

We use a new shifting technique to construct a basic and straightforward algorithm. This algorithm has a quadratic worst case time complexity though it performs well in practice. Using the shift tables of Knuth-Morris-Pratt algorithm [5] and Morris-Pratt algorithm [8], we make this first algorithm linear in the worst case. Then using a trie, we present a simple and very fast algorithm for small alphabets and long patterns.

We present the basic algorithm in Sect. 1. Section 2 is devoted to the linear variation of the basic algorithm. In the Sect. 3 we introduce the third algorithm which is very fast in practice. Results of experiments are given in Sect. 4.

2 The basic algorithm

2.1 Description

We want to solve the string matching problem which consists in finding all the occurrences of a pattern x of length m in a text y of length n. Both x and y are build over the same finite alphabet Σ of size σ. We are interested in the problem where the pattern x is given before the text y. In this case, x can be preprocessed to construct the data structure z.

The idea of our first algorithm is straightforward. For each symbol of the alphabet, a bucket collects all of that symbol's positions in x. When a symbol occurs k times in the pattern, there are k corresponding positions in the symbol's bucket. When the pattern is much shorter than the alphabet, many buckets are empty.

The main loop of the search phase consists of examining every mth text symbol, y_j (so there will be n/m main iterations). For y_j, use each position in the bucket $z[y_j]$ to obtain a possible starting point p of x in y. Perform a comparison of x to y beginning at position p, symbol by symbol, until there is a mismatch, or until all match.

The entire algorithm is given Fig. 1.

```
SkipSearch(x, m, y, n)
 1  ▷ Initialization
 2  for all symbols s in Σ
 3       do z[s] ← ∅
 4  ▷ Preprocessing phase
 5  for i ← 0 to m − 1
 6       do z[x_i] ← z[x_i] ∪ {i}
 7  ▷ Searching phase
 8  j ← m − 1
 9  while j < n
10       do for all i in z[y_j]
11               do if y_{j−i} ··· y_{j−i+m−1} = x
12                    then Report(j − i)
13          j ← j + m
```

Fig. 1. The basic algorithm.

2.2 Complexity

The space and time complexity of this preprocessing phase is in $O(m + \sigma)$. The worst case time complexity of this algorithm is $O(mn)$. This bound is tight and is reached when searching for $a^{m-1}b$ in a^n.

The expected time complexity when the pattern is small, $m < \sigma$, is $O(n/m)$. As m increases with respect to σ, the expected time rises to $O(n)$.

3 A linear algorithm

3.1 Description

It is possible to make the basic algorithm linear using the two shift tables of Morris-Pratt [8] and Knuth-Morris-Pratt [5].

For $1 \leq i \leq m$, $mpNext[i]$ = length of the longest border of $x_0x_1 \cdots x_{i-1}$ and $mpNext[0] = -1$.

For $1 \leq i < m$, $kmpNext[i]$ = length of the longest border of $x_0x_1 \cdots x_{i-1}$ followed by a character different from x_i, $kmpNext[0] = -1$ and $kmpNext[m] = m - period(x)$.

The lists in the buckets are explicitly stored in a table (see algorithm PreKmpSkipSearch in Fig. 3).

A general situation for an attempt during the searching phase is the following (see Fig. 2):

- j is the current text position;
- $i = z[y_j]$ thus $x_i = y_j$;
- $start = j - i$ is the possible starting position of an occurrence of x in y;
- $wall$ is the rightmost scanned text position;
- $y_{start}y_{start+1} \cdots y_{wall-1} = x_0x_1 \cdots x_{wall-start-1}$;
- $y_{wall} \neq x_{wall-start}$.

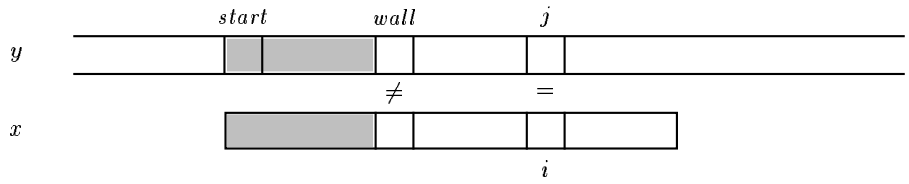

Fig. 2. General situation during the searching phase of the linear algorithm.

The comparisons will be performed from left to right between

$$y_{wall} y_{wall+1} \cdots y_{start+m-1}$$

and

$$x_{wall-start} x_{wall-start+1} \cdots x_{m-1}$$

Let $k \geq wall - start$ be the smallest integer such that $x_k \neq y_{start+k}$ or $k = m$ if an occurrence of x starts at position *start* in y.

Then *wall* takes the value of $start + k$.

Then compute two shifts (two new starting positions): the first one according to the skip algorithm (see algorithm AdvanceSkip in Fig. 5 for details), this gives us a starting position *skipStart*, the second one according to the shift table of Knuth-Morris-Pratt, which gives us another starting position *kmpStart*.

Several cases can arise:

case 1 $skipStart < kmpStart$ then a shift according to the skip algorithm is applied which gives a new value for *skipStart*, and we have to again compare *skipStart* and *kmpStart*;

case 2 $kmpStart < skipStart < wall$ then a shift according to the shift table of Morris-Pratt is applied which gives a new value for *kmpStart*, and we have to again compare *skipStart* and *kmpStart*;

case 3 $skipStart = kmpStart$ then another attempt can be performed with $start = skipStart$;

case 4 $kmpStart < wall < skipStart$ then another attempt can be performed with $start = skipStart$.

The complete algorithm is given in Fig. 6. When an occurrence of x is found in y, it is, of course, possible to shift by the length of the period of x.

3.2 Complexity

The time complexity of the second algorithm can be easily computed with the following arguments:

```
PRE-KMPSKIPSEARCH(x, m)
 1  ▷ Initialization
 2  for all symbols s in Σ
 3       do z[s] ← −1
 4  list[0] ← −1
 5  z[x[0]] ← 0
 6  for i ← 1 to m − 1
 7       do list[i] ← z[x[i]]
 8            z[x[i]] ← i
```

Fig. 3. Preprocessing of KMPSKIPSEARCH algorithm.

```
ATTEMPT(start, wall)
1  k ← wall − start
2  while k < m and x_k = y_{wall+k}
3       do k ← k + 1
4  return k
```

Fig. 4. Return $k \geq wall - start$ the smallest integer such that $x_k \neq y_{start+k}$ or $k = m$ if an occurrence of x starts at position $start$ in y when comparing $y_{wall} y_{wall+1} \cdots y_{start+m-1}$ and $x_{wall-start} x_{wall-start+1} \cdots x_{m-1}$.

```
ADVANCESKIP()
1  repeat j ← j + m
2    until j ≥ n or z[y_j] ≥ 0
3  if j < n
4    then i ← z[y_j]
```

Fig. 5. Compute the shift using the buckets.

```
KmpSkipSearch(x, m, y, n)
 1  wall ← 0
 2  i ← −1
 3  j ← −1
 4  AdvanceSkip()
 5  start ← j − i
 6  while start ≤ n − m
 7        do wall ← max(wall, start)
 8           k ← Attempt(start, wall)
 9           wall ← start + k
10           if k = m
11             then Report(start)
12                  i ← i − period(x)
13             else i ← list[i]
14           if i < 0
15             then AdvanceSkip()
16           skipStart ← j − i
17           kmpStart ← start + k − kmpNext[k]
18           k ← kmpNext[k]
19           while not (case 3 or case 4)
20  ▷ case 1       do if skipStart < kmpStart
21                       then i ← list[i]
22                            if i < 0
23                              then AdvanceSkip()
24                            skipStart ← j − i
25  ▷ case 2             else kmpStart ← kmpStart + k − mpNext[k]
26                            k ← mpNext[k]
27           start ← skipStart
```

Fig. 6. Searching phase of the KmpSkipSearch algorithm.

- during each attempt no comparison can be done to the left of the position *wall*, this implies that a text symbol can be compared positively only once. This gives n positive symbol comparisons;
- each negative symbol comparison implies a shift of length at least one, there can be at most $n - m + 1$ shifts;
- the total number of text symbols accessed to look into their bucket is $\lfloor n/m \rfloor$.

This gives us a total of $2n + \lfloor n/m \rfloor - m + 1$ symbol comparisons.

The expected time of the search phase of this algorithm is similar to the first algorithm.

4 A very fast practical algorithm

4.1 Description

Instead of having a bucket for each symbol of the alphabet. We can build a trie $T(x)$ of all the factors of the length $\ell = \log_\sigma m$ occurring in the pattern x. The leaves of $T(x)$ represent all the factors of length ℓ of x. There is then one bucket for each leaf of $T(x)$ in which is stored the list of positions where the factor, associated to the leaf, appears in x.

The searching phase consists in looking into the buckets of the text factors $y_j y_{j+1} \cdots y_{j+\ell-1}$ for all $j = k(m - \ell + 1) - 1$ with the integer k in the interval $[1, \lfloor (n - \ell)/m \rfloor]$.

The complete algorithm is shown in Fig. 7, it uses a function ADD-NODE given in Fig. 8.

4.2 Complexity

The construction of the trie $T(x)$ is linear in time and space in m providing that the alphabet size is constant, which is a reasonable assumption (see [7]). It is done using suffix links.

The time complexity of the searching phase of this algorithm is $O(mn)$. The expected search phase time is $O(\ell(n/(m - \ell)))$.

5 Experiments

We tested our three algorithms against three other, namely: the Boyer-Moore algorithm (BM) [1], the Tuned Boyer-Moore algorithm (TBM) [4] and the Reverse Factor algorithm (RF) [6] and [2]. We add to these algorithms a fast skip loop (*ufast* in the terminology of [4]) which consists in checking only if there is a match between the rightmost symbol of the pattern and its corresponding aligned symbol in the text before checking for a full match with all the other symbols. For the BM and TBM algorithms we also use Raita's trick [9] (already introduced in [5]) which consists in storing the first symbol of the pattern in a variable and checking if the leftmost symbol of the window match this variable before checking for a full match with all the other symbols.

```
AlphaSkipSearch(x, m, y, n)
 1  ▷ Initialization
 2  root ← Create-Node()
 3  suffix[root] ← ∅
 4  height[root] ← 0
 5  ▷ Preprocessing phase
 6  node ← root
 7  for i ← 0 to m − 1
 8       do if height[node] = ℓ
 9             then node ← suffix[node]
10          childNode ← child[node, x_i]
11          if childNode = ∅
12            then childNode ← Add-Node(node, x_i)
13          if height[childNode] = ℓ
14            then z[node] ← z[node] ∪ {i − ℓ − 1}
15          node ← childNode
16  ▷ Searching phase
17  j ← m − ℓ
18  while j < n − ℓ
19       do node ← root
20          for k ← 0 to ℓ − 1
21               do node ← child[node, y_{j+k}]
22          if node ≠ ∅
23            then for all i in z[node]
24                      do if y_{j−i} ··· y_{j−i+m−1} = x
25                            then Report(j − i)
26          j ← j + m − ℓ + 1
```

Fig. 7. AlphaSkipSearch algorithm.

```
Add-Node(node, s)
 1  childNode ← Create-Node()
 2  child[node, s] ← childNode
 3  height[childNode] ← height[node] + 1
 4  suffixNode ← suffix[node]
 5  if suffixNode = ∅
 6    then suffix[childNode] ← node
 7    else suffixChildNode ← child[suffixNode, s]
 8         if suffixChildNode = ∅
 9           then suffixChildNode ← Add-Node(suffixNode, s)
10         suffix[childNode] ← suffixChildNode
11  return childNode
```

Fig. 8. Add a new node labelled by s along the suffix path.

All these algorithms have been implemented in C in a homogeneous way such as to keep their comparison significant. The texts used are composed of 500000 symbols and were randomly built. The target machine is a Hyunday SPARC HWS-S310 running SunOS 4.1.3. The compiler is `gcc`.

For each pattern length, we searched for hundred patterns randomly chosen in the texts. The time given in the Table 1 are the times in seconds for searching hundred patterns.

Running times give a good idea of the efficiency of an algorithm but are very dependent of the target machine. We also present the number of inspections per text symbols (Table 2) which is a more theoretical measure.

Table 1. Running times for an alphabet of size 2.

algo.\m	10	20	40	80	160	320	640
BM	**38.05**	16.03	11.81	9.46	7.47	6.65	6.13
TBM	50.61	32.16	30.76	31.11	30.62	31.31	32.77
RF	42.55	**14.47**	**9.82**	**6.55**	**5.77**	6.91	10.77
SKIP	65.27	37.27	35.59	34.98	34.26	33.87	34.66
KMPSKIP	82.61	61.87	61.10	60.91	60.70	60.45	61.25
ALPHASKIP	54.63	19.01	11.69	8.07	5.87	**4.83**	**4.47**

Table 2. Number of inspections per symbol for an alphabet of size 2.

algo.\m	10	20	40	80	160	320	640
BM	0.6121	0.4505	0.3291	0.2700	0.2104	0.1815	0.1598
TBM	1.2369	1.2954	1.2536	1.2793	1.2499	1.2708	1.3114
RF	**0.5127**	**0.2942**	**0.1696**	**0.0970**	**0.0560**	**0.0338**	0.0238
SKIP	1.1980	1.0999	1.0502	1.0255	1.0138	1.0083	1.0087
KMPSKIP	0.9230	0.8604	0.8199	0.8028	0.7890	0.7931	0.7929
ALPHASKIP	0.7165	0.3897	0.2103	0.1141	0.0630	0.0361	**0.0211**

The tests show that the algorithm ALPHASKIP is the most efficient for searching long patterns, both practically and theoretically, when dealing with small alphabets (it is also the case for an alphabet of size 4).

6 Concluding Remarks

We presented a new string matching algorithm with an expected number of inspections of text symbols in $O(\log_\sigma m(n/(m-\log_\sigma m)))$ where m is the length of the pattern, n is the length of the text and σ is the size of the alphabet. The

algorithm uses a new shifting technique, based on collecting positions of short substrings of the pattern. The goal is to quickly discover whether and where the pattern x can occur in a window onto y. This algorithm performs well in both theory and practice.

This shifting technique can be extended for searching patterns with classes of symbols or for searching a finite set of patterns.

References

[1] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):762–772, 1977.

[2] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[3] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[4] A. Hume and D. M. Sunday. Fast string searching. *Software–Practice & Experience*, 21(11):1221–1248, 1991.

[5] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(1):323–350, 1977.

[6] T. Lecroq. A variation on the Boyer-Moore algorithm. *Theoret. Comput. Sci.*, 92(1):119–144, 1992.

[7] T. Lecroq. Experiments on string matching in memory structures. *Software–Practice & Experience*, 28(5):562–568, 1998.

[8] J. H. Morris, Jr and V. R. Pratt. A linear pattern-matching algorithm. Report 40, University of California, Berkeley, 1970.

[9] T. Raita. Tuning the Boyer-Moore-Horspool string searching algorithm. *Software–Practice & Experience*, 22(10):879–884, 1992.

[10] G. A. Stephen. *String searching algorithms*. World Scientific Press, 1994.